The war on error

James Chapman
University of Tartu

European Union
European Social Fund

Investing in your future

Syntax and basic definitions

Anatomy of a file

- A file usually starts with the first line
  - `module filename where`
- The rest of the file proceeds without indenting.
- Top level definitions must be declared before use.
- However, before the first line we an add options (usually to turn something off) which correspond to command line options:
  - `{-# OPTIONS --no-termination-check #-}`

Naming

- Agda is very liberal about naming:
  - There are a limited number of reserved characters: `@ . ( ) { } ; _` which cannot be used at all in names
  - and some reserved words which cannot be used in name parts:
    - `-> : = ? \ | → ∀ λ abstract data forall hiding import in infix infixl infixr let module mutual open postulate primitive private public record renaming rewrite using where with Prop Set[0-9]* [0-9]+`
- Names parts: Eg. `_part1_part2_` or `if_then_else_`

Spacing continued

- There are consequence of being liberal about naming.
  - Notice that `:  =  ?  \  |  →  ∀  λ` are reserved words not characters so they can be part of names:
    - Eg. `x+z=x`, `S→T`, `x:S` are valid names!
- Instead we must use spacing between different clauses `x : S` and `S → T` etc.
- Note that reserved characters do not need a space separator. Eg. `(x + y) + z, Module.function`

Comments

- single-line comments begin with `--`
  - Eg. `f x = ? -- I'll fill this in later`
- Multi-line comments begin with `{-` and end with `-}`
  - Eg. `{- The rest of this file is utterly incomprehensible -}`

Indenting

Indenting is used to structure the file into blocks like in Haskell, I recommend using two spaces each time. Eg.

```
data X : Set where
  con : X
```

and

```
f : X → X
f x = g x
  where
  g : X → X
  g x = x
```

Unlike Haskell this is the only option!

Unicode

- Unicode can be characters can be entered in TeX/LaTeX style. Eg. `\mu, \lambda, \Downarrow`
- Superscript ^ and subscript _ are proceeded by a \. Eg. `\pi\_0,  \sigma\^1`
- To see how to type a character in emacs place the cursor on it and type `C-u C-x =`
- Warning! It’s easy to get carried away with unicode but limited use of it is nice: greek symbols, and nice arrows.

Infix operators

- In Agda we can have mixfix operators Eg.
  - `_+_ : Nat → Nat → Nat`
  - `if_then_else_ : Bool → Bool → Bool → Bool`
- Type constructors, data constructors, functions can be named in this way.
- We can specify associativity and precedence for infix operators should be stated before use.
- `infixl 5 _*_`
- `infixr 6 _+_`
- `infix 6 _==_`

Local definitions

```
f : X → X
f x = g x
  where
  g : X → X
  g x = x
```

or

```
f : X → X
f x = g x
  where g : X → X
        g x = x
```

and

```
h : X → X
h x = let g : X → X
          g x = x
      in g x
  g x = x
```

Built-in types

You can declare Char, String, and Nat as built-in. Then you can use nicer syntax

```
postulate Char : Set
{-# BUILTIN CHAR  Char  #-}

postulate String : Set
{-# BUILTIN STRING String #-}

data Nat ...
{-# BUILTIN NATURAL Nat #-}
{-# BUILTIN ZERO    z   #-}
{-# BUILTIN SUC     s   #-}
```

λ-expression

The polymorphic identity function

```
id : {X : Set} → X → X

id x = x
```

can be rewritten as (the scope is extends as far as possible):

```
id = λ x → x
```

but we cannot pattern match on the RHS and there is not `case` expression like in Haskell

~~`f = λ (con x y) → x`~~

~~`f x = case x of (con y z) → x`~~

Note also that `let` *is a genuine expression and can go under a lambda but* `where` *is not and can't.*

Implicit arguments

Implicit arguments are stated in types with {} instead of ()

```
id : {X : Set} → X → X
id x = x
```

the type can sometimes be ommitted and inferred by Agda

```
id : ∀{X} → X → X
```

and the argument can be made explicit in the LHS

```
id {X} x = x
```

on the RHS any explicit argument can be left implicit by putting an _ but Agda might not know the answer!

```
id x = _
```

Inductive data types

```
data tcon1 : Set where
  dcon1 : tcon1
  dcon2 : tcon1 → tcon1

data tcon2 : Set where
  dcon3 : tcon2
  dcon4 : tcon2

data tcon3 : Set where
  dcon5 : tcon3

data tcon4 : Set where
```

What types are these?

Parameters and indices

```
data Vec (A : Set) : Nat → Set where
  []   : Vec A z
  _::_ : ∀{n} → A → Vec A n → Vec A (s n)
```

- Parameters come before the colon in the data declaration and are in scope of the constructors and don't vary.
- Indices come after and are not automatically in scope and can vary.

Postulates

- In Agda we can assume something without proof using a postulate
- We can delay a proof by just putting a ? but a postulate is taken as an axiom that we do not intend to prove later.
- Maybe we can't prove it at all: Eg.
  - `postulate EM : ∀(X : Set) → X ∨ ¬ X`

Records

We can define pairs in two ways in Agda. The first is the one you already know using a data type

```
data _×_ (A B : Set) : Set where
  _,_ : A → B → A × B
```

we could then define projections as follows

```
fst : {A B : Set} → A × B → A
fst (a , _) = a

snd : {A B : Set} → A × B) → B
snd (_ , b) = b
```

Records

The second is to use a record:

```
record _×_ (A B : Set) : Set where
  constructor _,_
  field       fst : A
              snd : B
open _×_ -- brings fst,snd into scope
```

We get projections automatically with the same name as the fields

```
 fst : {A B: Set} → A × B → A
```

The downside is we can’t pattern match on records.

Σ-type (dependent pair)

- I have already told you about the Π-type which the dependent function type.
- There is a dependent version of the pair type - the Σ-type. This can be defined as a record as follows:

```
record Σ (A :Set)(B : A → Set) where
  constructor _,_
  field       fst : A
              snd : B fst
```

It is the logical counterpart the "there exists" ∃

Σ-type (dependent pair)

Or as a data type

```
data Σ(A : Set)(B : A → Set) : Set where
  _,_ : (a : A) → B a → Σ A B
```

with projections

```
fst : ∀{A B} → Σ A B → A
fst (a , _) = a

snd : ∀{A B}(p : Σ A B) → B (fst p)
snd (_ , b) = b
```